Topological beam tracing

Steven Fortune[1]
Bell Laboratories

Abstract

We describe a topological beam-tracing algorithm that determines spatial visibility from a point source among a set of polygonal obstacles in $\mathbb{R}^3$. The algorithm uses an acyclic convex subdivision compatible with the set of obstacles. Under some assumptions, the running time is roughly proportional to the total combinatorial complexity of the intersection of the visible region with the convex subdivision. The algorithm has been implemented using exact arithmetic for the geometric predicates; implementation details and performance are presented.

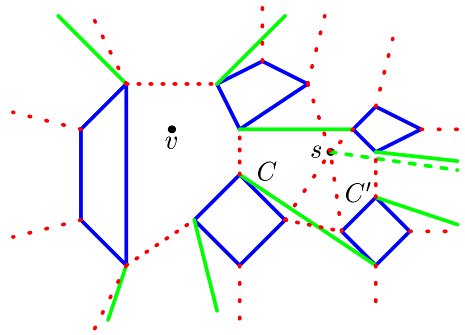

Figure 1: Topological and recursive beam tracing in 2d. Dark solid are obstacle boundaries; light solid are curtains, i.e., beam boundaries; dotted are subdivision edges.

1 Introduction

Given an environment in $\mathbb{R}^3$ consisting of obstacle polygons and a viewpoint p, a point q is *visible* if segment pq does not intersect any obstacles. The visible region or *beam* is the set of all visible points.

We give an algorithm that reports the beam from a viewpoint. The algorithm assumes an acyclic convex subdivision of space compatible with the obstacles. (See section 2.1.) A *chunk* is the intersection of the beam with a cell of the subdivision. Under reasonable assumptions about the subdivision, the running time of the algorithm is approximately proportional to the total number of vertices in all chunks. Of course, depending upon the subdivision, this number of vertices could be much larger than the number of vertices in the beam itself.

Sweeping a cell is the principal geometric subroutine of the beam-tracing algorithm. At some point the algorithm has determined the front of a chunk. (The *front* of a cell is the visible subset of the boundary of the cell, assuming the cell is opaque and there are no other obstacles; similarly the *back* is the invisible boundary subset.) To sweep a cell, the algorithm reports the chunk and then computes its back, pieces of which form pieces of the fronts of other chunks.

The visibility depth order on cells of the subdivision induces a partial order on the chunks of the beam. The algorithm sweeps the chunks of the beam consistently with the induced order. This ensures that the algorithm has enough information to report each chunk and to compute its back. By analogy with the topological sort of a graph, we call the algorithm *topological beam tracing*. The algorithm is described in more detail in section 2 below.

Recursive beam tracing. Algorithms for visibility and hidden-surface removal have been the subject of research in graphics and computational geometry for at least two decades. Various surveys are available [6, 10, 12, 16].

The algorithm that is closest in flavor to topological beam tracing is a recursive beam-tracing algorithm studied by Teller [15] and Funkhouser *et al* [10]. Their algorithm uses a procedure similar to sweeping at each cell. However, when a beam enters a cell through its front, the beam is intersected with each transparent back facet. Each beam fragment is then recursively propagated through the cell on the other side of the back facet.

[1]Room 2c459, Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ, USA, `sjf@research.bell-labs.com`.

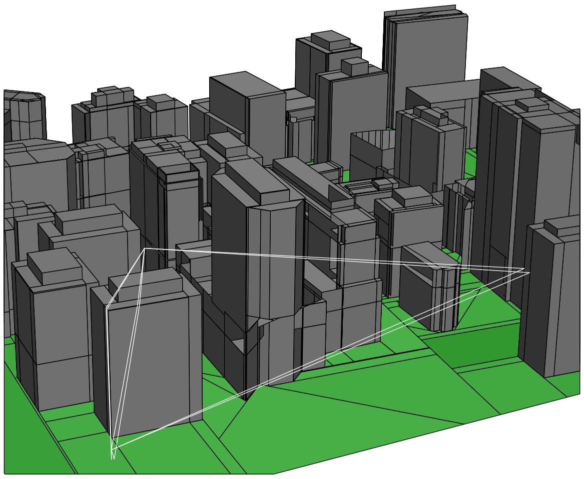

Figure 2: Propagation paths in the small Rosslyn model.

The difference between topological beam tracing and recursive beam tracing is illustrated in figure 1. Both algorithms sweep the cell containing v and then cell C in a similar fashion. However, recursive beam tracing splits the beam at vertex s, and propagates one fragment through each of the two transparent edges incident to s. The dashed line emanating from s is the common boundary of the two fragments. By contrast, topological beam tracing does not split the beam. The chunk in cell C' is only swept after both the chunks above and below s have been swept; this requires assembly of the chunk front. Clearly, beam fragmentation in two dimensions is caused by a *solitary* vertex such as s, i.e. a vertex with no opaque obstacle edges incident. It is easy to construct subdivisions that avoid solitary vertices.

In three dimensions, beam fragmentation is caused by a *solitary* edge, i.e. an edge with no opaque facets incident. In general it is impossible to avoid solitary edges completely. Teller and Funkhouser *et al* study the recursive beam tracing algorithm in highly occluded environments, such as the interior of an architectural building model. A good subdivision for such an environment is likely to have relatively few solitary edges, and beam fragmentation may be relatively unimportant. In unoccluded environments, such as the application described below, solitary edges are prevalent in any compatible subdivision, and topological beam tracing is advantageous.

Radio propagation. The beam-tracing algorithm forms the kernel of a ray-tracing system that simulates radio propagation in outdoor environments. Radio propagation is modelled using geometric optics and the uniform theory of diffraction. Geometric optics implies that propagation paths are straight, with specular reflection. Diffraction is modelled by a path that bends around a diffracting wedge (i.e. a corner), with diffraction point chosen so that the path is locally as short as possible.

A typical environment (figure 2 and section 5) covers a few square kilometers and has a few thousand polygons that represent both buildings and terrain. Topological beam tracing easily models geometric optics (with obvious modifications for beams created by reflection off obstacles). Diffraction is more complex, and is not discussed in this paper.

Implementation. Topological beam tracing is a reasonably complicated geometric algorithm with a subtle and distributed correctness invariant. It would be a considerable challenge to implement the algorithm reliably with floating-point arithmetic for the geometric primitives. Section 3 describes the bit-length requirements of the geometric primitives, assuming extended-precision integer arithmetic is used. (Other exact-arithmetic choices—sophisticated determinant-evaluation algorithms [2, 5], multi-component exact floating-point [13]—appear not to be adequate, in part because the predicates are required for constructed values.) Section 4 describes some implementation notes, and section 5 gives performance figures for the algorithm, using environments appropriate to the radio application.

2 Topological beam tracing

The *obstacle* set T is a properly intersecting set of polygons in $\mathbb{R}^3$. (A set of polygons is *properly intersecting* if the intersection of any pair of polygons is a vertex or edge of both.) A viewpoint p is just a distinguished point of $\mathbb{R}^3$. A point $q \in R^3$ is *visible* from viewpoint p if the interior of segment pq does not intersect any obstacle polygon. The *beam* from p, $B(p,T)$, is the closure of the set of visible points.

A point c is a *curtain* point if it is invisible and lies on the boundary of the beam; each curtain point c has a unique *shadow* point s so that s lies on segment pc, s is visible, and s lies on an obstacle. See figure 3. It is easy to check that s must lie on an obstacle edge e, and near s, all polygons incident to e must lie in one of the closed halfspaces of the plane through p and e. A *polygon shadow segment* S is a maximal connected set of shadow points contained in a single polygon edge; the *curtain* for S is the set of all curtain points whose shadow

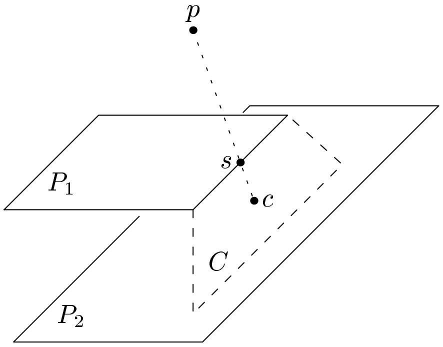

Figure 3: P_1 and P_2 are obstacle polygons; s is a shadow point for curtain point c; all of C is a curtain.

points belong to S. Clearly the beam is star-shaped from the viewpoint and its boundary consists of curtains and portions of obstacle polygons.

2.1 Convex subdivisions

Let S be a convex subset of $\mathbb{R}^3$. A finite partition $\mathcal{T}$ of S is a *(convex) subdivision* if each element $s \in \mathcal{T}$ is a relatively open convex polyhedron and each face of s is the union of some subset of $\mathcal{T}$. Elements of $\mathcal{T}$ of dimension 0, 1, 2, and 3 are *vertices*, *edges*, *facets*, and *cells*, respectively. A convex subdivision $\mathcal{T}$ is *compatible* with obstacle set T if each polygon in T is the union of some vertices, edges, and facets of $\mathcal{T}$. A facet of $\mathcal{T}$ is *opaque* if it is a subset of some polygon of T, otherwise it is *transparent.*

A point q on the boundary of cell C is a *front* point from a viewpoint p if the ray from p through q enters the interior of C just after q. Similarly q is a *back* point point if the ray leaves the interior of C just before q; q is a *silhouette* point if the ray avoids the interior of C. See figure 6. The *front* of a cell is all front points of the boundary of the cell; a *front* vertex, edge, facet, etc. is a subset of the cell front. Similar definitions hold for back and silhouette.

Let C and C' be cells of a subdivision $\mathcal{T}$. Then C *immediately precedes* C' from viewpoint p if C and C' are bounded by a common facet which is a back facet of C and a front facet of C'. The *(visibility) depth-order graph* of $\mathcal{T}$ from p is the directed graph whose nodes are the cells of $\mathcal{T}$ and with an arc from C to C' if C immediately precedes C'.

A subdivision is *acyclic* if the depth-order graph is acyclic from every viewpoint. Nondegenerate Delaunay triangulations [7] and the subdivisions that result from binary space partition trees [4] are acyclic.

2.2 Chunks

For the remainder of section 2, the viewpoint p, the obstacle set T, and an acyclic compatible subdivision $\mathcal{T}$ are fixed.

A *cell chunk* is a connected component of the intersection of beam $B(p, T)$ with a cell of the subdivision $\mathcal{T}$. See figure 6. Similarly, a *facet chunk* or an *edge chunk* is a connected component of the intersection of the beam with a subdivision facet or edge, respectively. The definitions of back and front extend to chunks in the obvious way.

A cell chunk C *immediately precedes* a cell chunk C' if C and C' are bounded by a common facet chunk that is a subset of the back of C' and the front of C. (Necessarily the common facet chunk is contained in a facet common to the two cells.) The *chunk-order graph* CG is the directed graph whose nodes are cell chunks and with a directed arc from chunk C to chunk C' if C immediately precedes C'.

Proposition 2.1 *The chunk-order graph is acyclic.*

A facet chunk is a polygon contained in the facet. It has a single outer boundary and possibly several inner boundaries surrounding holes. The outer boundary is a cycle consisting of edge chunks and *curtain edges*, which are the intersection of the facet with curtains of the beam. An inner boundary can consist only of curtain edges.

Consider an edge chunk e in the front of a cell C. Edge chunk e is a *silhouette* edge chunk if it is a subset of a silhouette edge of C. Clearly a silhouette edge chunk bounds exactly one facet chunk on the front of C. If e is not a silhouette edge but all its points are shadow points, then e is a *shadow* edge chunk. Again, a shadow edge chunk bounds exactly one facet chunk on the front of C. If e is neither silhouette nor shadow, then it is an *interior* edge chunk and bounds two facet chunks on the front of C.

The *front facet-chunk graph* FG is an undirected graph. The vertices of FG are facet chunks that are on the front of some cell chunk; two such facet chunks are connected by an edge in FG if they are on the front of the same cell chunk and they are bounded by a common interior edge chunk. Clearly, a connected component of FG corresponds to the front of a chunk.

2.3 Vertex-induced subgraphs

For G an undirected graph and a subset W of its vertices, the *vertex-induced subgraph* $G[W]$ is the undirected graph with vertex set W and with the edges of G that connect

```
// Preprocessing:
//      compute an acyclic subdivision
//      compatible with obstacle set T
// Input: viewpoint p
// Output: beam(p, T)
//
// SFG is a vertex-induced subgraph of FG
// R is a set of complete components of FG
proc beamtrace(p)
    SFG := ∅; R := ∅;
    Let C be the cell containing p
    for each facet f of C
        post(f)
    while R ≠ ∅
        Extract a cell-chunk front F from R
        Report(F)
        B := sweep(F)
        for each facet chunk f in B
            post(f)
    end while
end proc
```

Figure 4: Topological beam tracing. SFG, R and `sweep` are described in the text; `post` appears in figure 5.

two vertices in W. The *extension* of $G[W]$ by vertex v is $G[W \cup \{v\}]$. A connected component of $G[W]$ is *complete* if it is identical to a connected component of G. The connected component containing v is denoted $\mathrm{comp}(v)$.

2.4 The algorithm

The beam-tracing algorithm is presented in figure 4. It uses two auxiliary procedures `post` and `sweep`. Procedure `post` appears in figure 5. Procedure `sweep` computes the back of a cell chunk from the front of the cell chunk; an implementation is described in more detail below. As presented, the algorithm does not explicitly construct the beam; rather, it reports the front of each cell chunk one at a time.

Two data structures are used to determine when a cell chunk can be swept (i.e. when procedure `sweep` can be called for the cell chunk front). Graph SFG is a vertex-induced subgraph of the front facet-chunk graph FG. R is the *ready* set of complete components of FG. A *frontier* facet chunk is one that is a subset of both the back of a swept cell and the front of an unswept cell. Let W be the set of frontier facet chunks at the start of some iteration of the **while** loop in figure 4. It is easy to check that the algorithm maintains the invariant that $SFG \cup R = FG[W]$, where R is interpreted as a graph.

```
proc post(f)
    if f is not opaque
    then extend SFG by f
        if comp(f) is complete
        then R := R ∪ {comp(f)}
            SFG := SFG \ {comp(f) }
        end if
    end if
end proc
```

Figure 5: Posting a facet chunk; f is a facet chunk. R and SFG are as in figure 4. The test for completeness is discussed in section 2.6.

2.5 Time complexity

Subdivision $\mathcal{T}$ has *constant complexity cells* if the number of facets of each cell is bounded by a constant. Let b_c be the number of vertices of cell chunk c. Set $b_{\max} = \max\{b_c\}$ and $b = \sum b_c$, where c varies over all cell chunks of $\mathrm{beam}(p, T)$.

Theorem 2.2 *Assuming that $\mathcal{T}$ has constant complexity cells, procedure* `beamtrace` *can be implemented to report all chunks of beam(p, T) in time $O(b \log b_{\max})$.*

Proof: Straightforward, in the full paper. □

2.6 Avoiding SFG

The `beamtrace` implementation implied by theorem 2.2 requires that the internal graph structure of SFG be maintained. In practice this is somewhat complex, in effect requiring dynamic updates to a two-dimensional subdivision. The next two sections describe implementations of `post` and `sweep` that instead maintain one-dimensional curtain-edge chains. These implementations are asymptotically less efficient, but are simpler and probably faster in practice.

Consider a cell chunk C' contained in cell C. The front of C consists of vertices, edges, and facets forming a two-dimensional subdivision bounded by the silhouette edges of C. The front of C' consists of vertices, edge chunks, and facet chunks; its boundary consists of shadow edge chunks and curtain edges in addition to silhouette edge chunks. For the following, we can treat a shadow edge chunk as a curtain edge, since it is in fact the start of a beam curtain. A maximal connected subset of curtain edges is either a chain from silhouette edge to silhouette edge or a cycle, either inner or outer.

Suppose e is an interior edge chunk on the front of cell C; then it is incident to two facet chunks on the front of C. If exactly one of the facet chunks is a frontier chunk, then e is a *dangling* edge chunk.

The representation of a component of SFG is its set of curtain-edge chains and cycles. If the component is complete, any chain necessarily extends from silhouette to silhouette; if the component is not complete, then a chain may terminate at a dangling edge chunk as well.

The components of SFG and function comp can easily be maintained with the classic union-find data structure [1]. Though we do not give details, it is not hard to maintain the chains and cycles of components of SFG as they are merged together.

2.7 Procedure `sweep`

Let F be the front and B the back of a cell chunk contained in cell C. Clearly, the boundary of B consists of the silhouette edge chunks from F and for each curtain edge e of F, the central projection from v of e on the back of C. See figure 6.

The first step of `sweep` is to project all curtain-edge chains and cycles on B. A curtain-edge chain κ on the front of C starts and ends at points on silhouette edges of C. The projection of κ on the back of C starts and ends at the same points, and can be found by a simple procedure that traces the projection while stepping through the facets of the back of C. A similar procedure traces the projection of a curtain-edge cycle, after first locating the projection of a curtain-edge vertex on the back of C.

The second step is to assemble the back facet chunks. A *partial* facet chunk is a proper subset of its facet; the partial facet chunks can be found as follows. For each back facet f labelled with a curtain edge, sort the endpoints of the curtain-edge chains that traverse f in cyclic order around the boundary of f. From the sort, the outer boundaries of all the facet chunks contained in f can be found. Inner cycles, if any, can be assigned to the appropriate facet chunks, for example using a primitive that determines if two cycles nest.

A *full* facet chunk is a whole facet; similarly a *full* edge chunk is a whole edge. A full facet chunk has two properties: it is not labelled with any curtain edges, and it is bounded only by full edge chunks. Hence any unlabelled facet on the back of C that is incident to a partial facet chunk across a full edge chunk is a full facet chunk. Furthermore, any unlabelled facet on the back of C incident to a full facet chunk is itself a full facet chunk. Hence the full facet chunks can be found by a graph search through the back of C, starting at all full edge chunks contained in partial facet chunks.

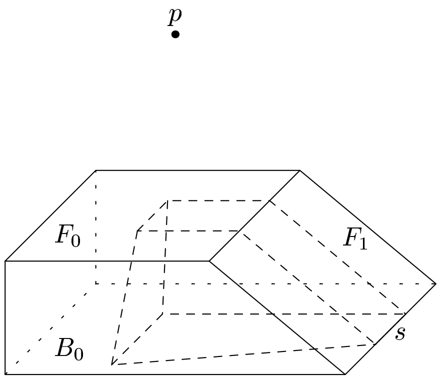

Figure 6: Point p is the viewpoint; facets F_0 and F_1 are part of the cell front; facet B_0 is part of the cell back; s is a silhouette edge chunk, part of a silhouette edge. The chunk outline is dashed.

2.8 Procedure `post`

Let f be a transparent back facet chunk discovered by `sweep`. Facet chunk f lies in the front of some cell C; `post` must classify each edge chunk e bounding f as silhouette, shadow, or interior, with f viewed as a front chunk of C. It is not hard to check that if e is not a silhouette edge chunk of C, then e is a shadow edge chunk if there is some opaque subdivision facet incident to the edge containing e, and e is an interior edge chunk otherwise.

Procedure `post` maintains a map M from dangling edge chunks to components; $M[e]$ is always the frontier facet chunk bounded by e. To extend SFG by a new facet chunk f, `post` creates a new component for f. For each interior edge chunk e bounding f, the algorithm queries $M[e]$. If $M[e]$ is defined, then `post` merges comp($M[e]$) with comp(f), and removes e from the domain of M. If $M[e]$ is undefined, then $M[e]$ is set to comp(f). Notice procedure `post` does nothing for silhouette and shadow edge chunks.

To test if a component is complete, `post` maintains with each component C the number of its dangling edge chunks, i.e., the number of edge chunks e in the domain of M with comp($M[e]$) $= C$. This count is easily updated when components are created and merged as described above. If the number of dangling edge chunks reaches 0, then the component is complete.

An edge chunk can be identified by the subdivision edge which contains it and by its two endpoints, each of which is either a subdivision vertex or the intersection of the edge with a beam curtain. Potentially there can be many edge chunks per edge. The mapping M could be implemented with a balanced tree, which guarantees a logarithmic worst-case query time. It is adequate in practice to implement M with a hash function indexed by the subdivision edge containing the edge chunk.

2.9 Reflection

A reflected beam is determined from a visible subpolygon P of an obstacle B. The reflected viewpoint r is the reflection in the obstacle plane of the original viewpoint. The reflected beam consists of all points visible from r through P after P, i.e. all points q so that segment rq intersects P at a point p and no obstacle intersects the interior of pq. The reflected beam has an outer boundary that consists of all points visible through an edge of P after the edge.

The construction of the compatible subdivision may have resulted in the partition of obstacle B into multiple facets. Hence P is partitioned into *(visible polygon) chunks*, where each such chunk is the intersection of P with one of the facets comprising B. Since tracing of the original beam reports visibility facet by facet, P is reported a chunk at a time. A process entirely analogous to the discovery of ready components of SFG is required to discover when P is complete.

Procedure `beamtrace` extends readily to reflected beams. A facet of the outer boundary can be treated as a curtain simply by treating the corresponding edge of P as a shadow edge. To initialize the ready set R of complete chunk fronts, procedure `post` is called for each chunk of P.

Section 4.3 below discusses modifications required to prevent coordinate bit-length growth.

3 Geometric primitives

This section describes the geometric primitives needed for `beamtrace`, emphasizing the requisite arithmetic bit-length assuming integer coordinates.

Procedure `beamtrace` requires an acyclic convex subdivision compatible with the obstacles. Typically, obstacle coordinates are given in floating-point and are only approximately known; furthermore the obstacles are “polygon soup”, i.e. have unintended intersections and gaps. Hence it is appropriate to perturb the obstacles to clean them up; it is also reasonable to choose obstacle coordinates so as to minimize the arithmetic bit-length required for subsequent geometric predicates. The perturbed data is *surrogate* data; it serves as a surrogate for the true input data during computation.

There are two natural choices for a bounded bit-length representation of a convex subdivision. The *vertex-based* representation has an *a priori* bound on vertex coordinate bit-length; facet plane coordinates are then inferred from the vertex coordinates (and have about three times the bit-length). The *facet-based* representation has an *a priori* bound on facet plane coordinates; vertex coordinates are inferred from facet planes (and again have about three times the bit-length).

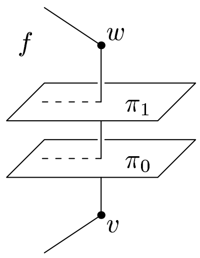

Figure 7: Curtain plane sorting.

The only known general way of producing an acyclic three-dimensional subdivision compatible with obstacles is with binary-space partition (BSP) trees [4]. Clearly, if the partitioning planes have bounded bit-length coordinates, the resulting subdivision is facet-based. This implies that the implementation of `beamtrace` should use a facet-based representation of the subdivision. Furthermore, the input obstacles should be converted to a surrogate representation based on planes with bounded bit-length (see section 4.1).

3.1 Sample geometric primitives

The following is a list of some of the primitives required by `beamtrace`. The list is not comprehensive, but does include the most complex primitives.

1. Curtain plane construction: construct the plane through the viewpoint and a subdivision edge.
2. Vertex/curtain plane sidedness: classify a subdivision vertex with respect to a curtain plane (above, on, or below).
3. Intersection point/curtain plane sidedness: classify the intersection point of a curtain plane and a subdivision edge with respect to another curtain plane. See figure 7.
4. Curtain line/subdivision line orientation: determine the orientation of the line through a curtain ray with respect to the line through a subdivision edge. See figure 8.

The third primitive is used in the assembly of facet chunks (section 2.7). In figure 7, two curtains c_0 and c_1

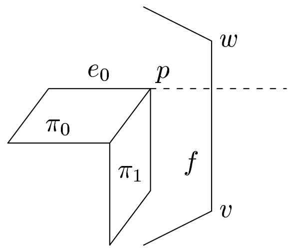

Figure 8: Curtain ray/subdivision edge orientation.

with curtain planes π_0 and π_1 intersect facet f in two curtain edges (shown dashed), both incident to edge vw. The curtain edges need to be sorted along vw. This can be done by classifying $\pi_0 \cap vw$ with respect to π_1.

The fourth primitive is used to decide if a curtain edge meets the edge of a facet or ends in the interior of a facet (section 2.7). See figure 8. Suppose two curtains c_0 and c_1 with planes π_0 and π_1 are incident, and c_0 is known to intersect the interior of a facet f in an edge e_0. Using the vertex-curtain plane sidedness test, it is possible to find edge vw bounding f that is intersected by π_0. Let π_f be the plane through f. Then a line orientation test with line $\pi_0 \cap \pi_1$ and line vw decides whether $p = \pi_0 \cap \pi_1 \cap \pi_f$ lies in the interior of f, on edge vw, or outside f.

3.2 Primitive bit-length

Let α ('approximation') and β ('bounding box') be positive integers and let $B \subset \mathbb{R}^3$ be the cube consisting of all points whose coordinates have absolute value at most 2^β.

The homogeneous representation (x, y, z, w) (of the point $(x/w, y/w, z/w)$) has *bit-length* (α, β) if x, y, z, w are all integers with $|x|, |y|, |z| < 2^{\alpha+\beta}$ and $|w| < 2^\alpha$. The representation (a, b, c, d) (of the plane $\{(x, y, z) : ax + by + cz + d = 0\}$) has *bit-length* (α, β) if a, b, c, d are integers with $|a|, |b|, |c| < 2^\alpha$ and $|d| < 2^{\alpha+\beta}$. The (α, β) bit-length bound also extends to lines represented in Plücker coordinates (described in the full paper).

Given an arbitrary point $p \in B$, there is an approximating point p' within distance $O(2^{-\alpha})$ of p so that p' has a homogeneous representation of bit-length (α, β). Similarly, given an arbitrary plane π intersecting B and a point $p \in \pi \cap B$, there is an approximating plane π' and a point $p' \in \pi'$ so that π' has representation bit-length (α, β), the unit normals of π and π' differ by $O(2^{-\alpha})$, and the distance from p to p' is $O(2^{-\alpha})$ [8]. (Lattice-basis reduction [11] often offers a better approximation within the same bit-length bound, but at considerable computational cost.)

Figure 9 gives bit-length bounds for some standard constructors and predicates [14], omitting small additive constants. The value of β is assumed to be the same for each argument of an operation, and is the same for each result, to within a small additive constant. (There is no requirement that the result meet the bounding box B.) Figure 10 gives bit-length estimates for primitives (1) through (4) above in the facet-based subdivision representation, assuming that each facet-plane has representation bit-length (α, β). For reference, the bit-length estimates are given for the vertex-based representation, assuming vertices have bit-length (α, β). In either case, the viewpoint is assumed to have representation bit-length (α, β).

Arguments	result	bit-length bound
P × P	L	$(\sum \alpha_i + \beta, \beta)$
P × P × P	π	$(\sum \alpha_i + 2\beta, \beta)$
P × L	π	$(\sum \alpha_i + \beta, \beta)$
$\pi \times \pi$	L	$(\sum \alpha_i, \beta)$
$\pi \times \pi \times \pi$	P	$(\sum \alpha_i, \beta)$
π × L	P	$(\sum \alpha_i, \beta)$
P × π	sign	$\sum \alpha_i + \beta$
P × P × P × P	sign	$\sum \alpha_i + 3\beta$
$\pi \times \pi \times \pi \times \pi$	sign	$\sum \alpha_i$
L × L	sign	$\sum \alpha_i + \beta$

Figure 9: Bit-length bounds for constructors (first 6) and orientation predicates (last 4). The arguments and results are points (P), lines (L), planes (π), or sign values $(-1, 0, +1)$; the ith argument has bit-length bound (α_i, β).

Primitive or object	Vertex	Facet
Viewpoint	(α, β)	(α, β)
Subd. vertex (SV)	(α, β)	$(3\alpha, \beta)$
Subd. line (SL)	$(2\alpha + \beta, \beta)$	$(2\alpha, \beta)$
Subd. plane (Sπ)	$(3\alpha + 2\beta, \beta)$	(α, β)
Curtain plane (Cπ))	$(3\alpha + 2\beta, \beta)$	$(3\alpha + \beta, \beta)$
Curtain line (CL)	$(6\alpha + 4\beta, \beta)$	$(6\alpha + 2\beta, \beta)$
Inter. point (IP)	$(5\alpha + 3\beta, \beta)$	$(5\alpha + \beta, \beta)$
Cπ/SV sidedness	$4\alpha + 3\beta$	$6\alpha + 2\beta$
CL/SL orientation	$8\alpha + 6\beta$	$8\alpha + 3\beta$
IP/Cπ sidedness	$8\alpha + 6\beta$	$8\alpha + 3\beta$

Figure 10: The bit-length required by some geometric primitives, for both vertex- and facet-based convex subdivisions. 'Curtain plane' is primitive 1 in section 3.1; the last three lines are primitives 2 through 4.

4 Implementation

The `beamtrace` algorithm has been coded in C++. It requires about 800 lines for the geometry primitives, 1200 lines for representing and building the convex subdivision, 1300 lines for polygon modeling, and 1800 lines for the topological beam tracing itself. The radio application is several thousand lines more. BSP trees are created using a variant of Teller's heuristic [15]. Bit-length $(\alpha, \beta) = (20, 20)$ is used for the subdivision and viewpoint. The maximum required arithmetic bit-length is about 230 bits ($\approx 8\alpha + 3\beta$).

4.1 Surrogate polygons

The *plane representation* of a polygon consists of a *supporting plane* σ and a vector of *edge planes* $\pi_0, \ldots, \pi_k$. It represents the polygon contained in σ with vertices $\sigma \cap \pi_i \cap \pi_{i+1}$, $i = 0, \ldots, k$ (with addition taken mod $k+1$). A plane representation has bit-length (α, β) if each plane has bit-length (α, β).

In the radio application, the obstacles consist of building walls and terrain. Building walls are polygons, usually vertical (parallel to the z-axis) or horizontal. Terrain is a triangulated surface (obtained from a height field). The conversion from input to surrogate polygons in the plane representation is a bit involved and is different in the two cases.

The conversion of building walls to surrogate polygons also attempts to clean up the input and is briefly described as follows. First, nearby vertices are collapsed into clusters. An approximation of bit-length $(\alpha, \beta) = (20, 20)$ to the supporting plane of each input polygon is determined. Then for each edge of each input polygon, a corresponding edge plane of the surrogate polygon is chosen. Preferably the edge plane is the approximate supporting plane of another polygon sharing the edge (found using vertex clusters and hashing). If there is no such polygon, the edge plane is chosen as approximately perpendicular to the supporting plane.

Terrain vertices typically have more than three terrain triangles incident (building vertices usually have only three distinct wall planes incident). If the surrogate planes of the triangles incident to an input vertex are found independently, then the planes in general will not all meet at a common point. The construction of the subdivision using a BSP tree will still succeed, but the input vertex will be replaced by a complicated structure containing many intersection points of triples of surrogate planes. This is clearly undesirable, since the size of the subdivision is increased.

This problem can be avoided by restricting terrain triangles. Each triangle vertex must have integral coordinates less than $2^\alpha = 2^{20}$, and the L_∞ length of each edge must be less than $2^{\beta/2} = 2^{10}$. The coefficients of the plane through a terrain triangle $\Delta p_1 p_2 p_3$, $p_i = (x_i, y_i, z_i)$, are the 3×3 minors of

$$\begin{bmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 & 0 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 & 0 \end{bmatrix}.$$

It is easy to verify that the resulting plane has representation bit-length $(\alpha, \beta) = (20, 20)$. Hence there is no need to replace input triangles with surrogate triangles. The restriction on input vertices can be enforced in the algorithm that produces a triangulated surface from a height field; the restriction usually does not significantly increase the approximation error introduced by the algorithm.

4.2 Arithmetic

The geometric primitives were implemented using LN [9]. LN provides efficient multiprecision integer arithmetic together with a floating-point filter to speed predicate evaluation. The floating-point filter uses static error bounds, which are cheap but too conservative in this application. Less conservative filters are known (e.g. semistatic filters [3]), but are much more expensive, typically 2 or 3 times the cost of actual floating-point evaluation.

A filter that is both cheap and effective results by distinguishing coordinates that usually have their full static bit-length from coordinates that are more likely to have variable bit-length. Consider a plane representation (a, b, c, d) with bit-length (α, β) (so $|a|, |b|, |c| < 2^\alpha$ and $|d| < 2^{\alpha+\beta}$). It is reasonable to assume that $2^{\alpha-1} \le \max(|a|, |b|, |c|) < 2^\alpha$, because coordinates can be uniformly multiplied by a constant. The orientation test on four such planes $\pi_i = (a_i, b_i, c_i, d_i)$, $i = 1, \ldots, 4$, is the sign of

$$\begin{vmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{vmatrix}.$$

A good error bound for the floating-point evaluation of this expression is

$$\kappa \epsilon 2^{3\alpha} (|d_0| + |d_1| + |d_2| + |d_3|),$$

where κ is a constant and ϵ is floating-point precision. By caching $|d|$ with each plane, the error bound computation requires three additions and a multiplication.

This error bound can be extended to the orientation test on planes with distinct bit-lengths (α_i, β), by caching with each plane a floating-point approximation

Data set	size	poly	subdivision		beamtrace			
	km × km		cells	time	time	chunks	edges	opaque
Rosslyn (small)	.5 × 0.4	943	2234	2.2	.10	576	3365	369
Rosslyn (large)	.5 × 1.3	1564	5806	5.2	.12	633	3267	311
Manhattan (small)	1.0 × 1.5	3658	4546	9.4	.28	1795	12306	376
Manhattan (large)	6.0 × 3.0	27438	33604	127.9	7.50	32797	247985	7601

Figure 11: Executions of `beamtrace`, single point source. Columns are the approximate size and number of polygons in the environment, the number of cells and the time to build the subdivision (in seconds), the time for `beamtrace` (seconds), the number of chunks in the beam, the total number of curtain edges on all chunk fronts, and the number of curtain edges on opaque facets.

$(\hat{a}_i, \hat{b}_i, \hat{c}_i, \hat{d}_i)$ scaled so that $\max(|\hat{a}_i, \hat{b}_i, \hat{c}_i|)$ is approximately uniform (say $\approx 2^\alpha$). The floating-point filter uses the scaled approximations to evaluate both orientation and error bound; a slightly larger value of κ (κ=42) accommodates the error in the approximations.

The LN-generated predicates were extended with (hand-coded) filters created using this technique. Empirically, the filter is quite effective: for a typical instance of `beamtrace`, with about 20 million predicate evaluations, some 10% required exact evaluation. But of those, only a very small number–about 100–were not zero.

4.3 Reflection

A reflected beam is determined from a visible subpolygon P of an obstacle polygon (see section 2.9). Consider the curtain c of the unreflected beam incident to an edge e of P, and the corresponding outer boundary facet b of the reflected beam incident to e. The plane through b is the reflection in the plane of the obstacle of the plane through c. If computed exactly, it would have more than double the bit-length of the curtain plane through b. Since reflection can nest arbitrarily, exact computation of reflecting beams leads to unbounded (in fact exponential) bit-length growth.

Coordinate bit-length growth can be avoided by replacing the true reflected beam with an approximation. The true reflected viewpoint is replaced with an approximating point $\tilde{p}$ with bit-length (α, β) (the same as the original viewpoint); P is replaced with an approximating polygon $\tilde{P}$ with bit-length (α, β) in the plane representation. The approximating beam is defined as all points visible from $\tilde{p}$ through $\tilde{P}$ after $\tilde{P}$. It is easy to verify that the outer-boundary facet planes of the approximating beam have the same bit length as curtain planes.

The computation of $\tilde{P}$ is not completely straightforward. Polygon P is already essentially in the plane representation, with edges defined by curtain planes of the unreflected beam. A polygon $\hat{P}$ can be obtained by rounding each curtain plane to bit-length (α, β); however $\hat{P}$ may have self-intersections, since edges are perturbed by rounding. Instead $\tilde{P}$ is defined as the points of the base plane of P that have positive winding number with respect to $\hat{P}$, viewing $\hat{P}$ as a closed cycle [8]. Then $\tilde{P}$ is a simple polygon, perhaps with holes or multiple components.

5 Performance

We give a brief indication of performance; all times are on an SGI MIPS R10000 running at 250 Mhz. Typically 20% of runtime is devoted to (multiprecision) exact arithmetic, and about 13% to floating-point arithmetic (including filters).

Figure 11 gives typical performance for point-source visibility in several urban and suburban environments. The 'small' Rosslyn data set (figure 2) and the two Manhattan data sets include only building walls (vertical and horizontal). The 'large' Rosslyn data set also include terrain triangles at many orientations; this considerably increases the relative number of cells in the subdivision. The total number of curtain edges measures the work done by `beamtrace`, while the number of curtain edges on opaque facets measures the size of the output, i.e. of the beam itself. The ratio gives an indication of the overhead imposed by the subdivision; it varies widely with the data set.

The implementation is not optimized to reduce storage; for example, the large Manhattan example requires about 300 megabytes of memory. Caching and paging overhead appear to be significant. Memory use could be reduced, perhaps by a factor of two or more, by compact representations of extended-precision coordinates and of the subdivision itself.

In the radio application it is important to include reflected beams as well. Figure 12 gives performance as a function of the number of reflections. Clearly, the time

Refl	time	beams	chunks	edges	opaque
1	.7	231	4525	27648	3046
2	2.7	1327	16574	98532	11652
3	6.6	4324	40972	238657	30643
4	12.0	10310	81480	465464	64268
5	22.7	20401	143414	805888	117082

Figure 12: Performance on small Rosslyn data set, as a function of the number of reflections.

Data set	Refl	Topological	Recursive
Rosslyn (small)	0	576	27020
	5	143414	3093312
Manhattan (small)	0	1795	70160
	3	34806	778279
Interior	0	127	1126
	5	35325	272994

Figure 13: Number of chunks for topological and recursive beam tracing. 'Interior' is the interior of a small building model with 276 polygons.

per beam decreases with the number of reflections; this is because the average angular width of a beam decreases with the number of reflections. The ratio of total curtain edges to curtain edges on opaque facets decreases with the number of reflections.

It is easy to simulate recursive beam tracing with `beamtrace` (by treating an interior edge chunk as a shadow edge chunk). Figure 13 compares the number of chunks required by topological and recursive beam tracing. For unoccluded models like Manhattan and Rosslyn, recursive beam tracing requires 20-40 times as many chunks as topological beam tracing. For an occluded model (the interior of a small building), topological beam tracing is still advantageous, though less significantly. The relative advantage of topological beam tracing decreases with the number of reflections, probably because of the reduced angular width of reflected beams.

References

[1] A. Aho, J. Hopcroft, J. Ullmann, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[2] H. Brönnimann, M. Yvinec, Efficient exact evaluation of signs of determinants, to appear, *Algorithmica.*

[3] C. Burnikel, S. Funke, M. Seel, Exact geometric predicates using cascaded computation, *Proc. Fourteenth Ann. Symp. Comp. Geom*, 175–183, 1998.

[4] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry: Algorithms and applications*, Springer-Verlag, 1997.

[5] K. L. Clarkson, Safe and efficient determinant evaluation, *Proc. 33rd Ann. IEEE Symp. Found. Comp. Science*, 387–395, 1992.

[6] D. Dobkin, S. Teller, Computer graphics, in *Handbook of Discrete and Computational Geometry*, 779–795, J. E. Goodman, J. O'Rourke, eds., CRC, 1997.

[7] H. Edelsbrunner, An acyclicity theorem for cell complexes in d-dimensions, *Combinatorica*, 10:251–260, 1990.

[8] S. Fortune, Polyhedral modelling with multiprecision integer arithmetic, *Computer-Aided Design*, 29(2):123–133, 1997.

[9] S. Fortune, C. Van Wyk, Static analysis yields efficient exact integer arithmetic for computational geometry, *ACM Trans. Graphics*, 15(3):223–248, July 1996.

[10] T. Funkhouser, I. Carlbom, G. Elko, G. Pingali, M. Sondhi, J. West, A beam-tracing approach to acoustic modeling for interactive virtual environments, *SIGGRAPH98 Conference Proceedings*, 21–32, 1998.

[11] L. Lovasz, *An Algorithmic Theory of Numbers, Graphs, and Convexity*, SIAM, 1986.

[12] J. O'Rourke, Visibility, in *Handbook of Discrete and Computational Geometry*, 467–479, J. E. Goodman, J. O'Rourke, eds., CRC, 1997.

[13] J. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, *Disc. Comp. Geom*, **18**(3), 303–363, 1997.

[14] J. Stolfi, *Primitives for computational geometry*, Ph.D. Thesis, Department of Computer Science, Stanford University, 1988.

[15] S. Teller, Visibility computations in densely occluded polyhedral environments, Ph.D. Thesis, U.C. Berkeley, 1992.

[16] A. Watt, M. Watt, *Advanced animation and rendering techniques: theory and practice*, Addison-Wesley, 1992.